ARL-TR-7767 ● AUG 2016

US Army Research Laboratory

Industrial Control System Process-Oriented Intrusion Detection (iPoid) Algorithm

by Daniel T Sullivan, Edward J Colbert, Kenneth D Renard, Phillip L Tucker, Travis W Parker, Stephen R Neyens, and Christopher A Walsh

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

ARL-TR-7767 ● AUG 2016

US Army Research Laboratory

Industrial Control System Process-Oriented Intrusion Detection (iPoid) Algorithm

by Daniel T Sullivan, Edward J Colbert, Kenneth D Renard, Phillip L Tucker, Travis W Parker, Stephen R Neyens, and Christopher A Walsh

Computational and Information Sciences Directorate, ARL

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED (From - To)
August 2016	Technical Report	July 2014–June 2016

4. TITLE AND SUBTITLE
Industrial Control System Process-Oriented Intrusion Detection (iPoid) Algorithm

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Daniel T Sullivan, Edward J Colbert, Kenneth D Renard, Phillip L Tucker, Travis W Parker, Stephen R Neyens, and Christopher A Walsh

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
US Army Research Laboratory
ATTN: RDRL-CIN-S
2800 Powder Mill Road
Adelphi, MD 20783-1138

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-TR-7767

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
This report describes the software architecture and capabilities of an industrial control system process-oriented intrusion detection (iPoid) algorithm developed in the Army Cyber-Research Analytics Laboratory (ACAL) at the US Army Research Laboratory. The iPoid algorithm performs packet inspection of Modbus transmission control protocol communications by applying rules to detect suspicious activity. ACAL's iPoid creates alert messages for security analysts if further investigation is required. We illustrate the iPoid algorithm using a research intrusion-detection system. This report describes the iPoid algorithm and how its software functions, how to write the analysis rules, and how to test the software.

15. SUBJECT TERMS
supervisory control and data acquisition (SCADA), Modbus, industrial control system, intrusion detection system

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Daniel T Sullivan
Unclassified	Unclassified	Unclassified	UU	34	19b. TELEPHONE NUMBER (Include area code) 301-394-0248

Contents

List of Figures

List of Tables

Acknowledgments

We appreciate Dr Alexander Kott's and Mr Curtis Arnold's support of Industrial Control Systems–Supervisory Control and Data Acquisition research at the US Army Research Laboratory.

1. Background

An industrial control system (ICS) manages automated processes in multiple sectors of the global economy. ICSs can be found in manufacturing plants, transportation systems, food and medicine production, and critical infrastructure. An ICS may be local to one factory or it may control automated processes across thousands of miles as in the energy sector. The key discriminator separating an ICS from an information technology (IT) system is that an ICS monitors or interacts with something physical in the real world. Previously, ICSs were isolated networks; however, due to a demand for greater productivity and efficiency, IT and ICS networks are being interconnected. This may expose an ICS network to the same threats as a corporate IT network.[1]

In Fig. 1 we depict a typical manufacturing plant with a corporate IT network and an ICS network. The ICS network consists of the Supervisory Control and Basic Control layers and any field bus networks connected to plant sensors or actuators. In accordance with best practices, an Operations demilitarized zone (DMZ) protects the ICS from external network threats. In this design, the ICS and corporate networks do not share resources to minimize intrusion risks. Each network will have its own Active Directory, human–machine interface (HMI), Dynamic Host Configuration Protocol (DHCP), and Domain Name System (DNS) servers to provide these services.

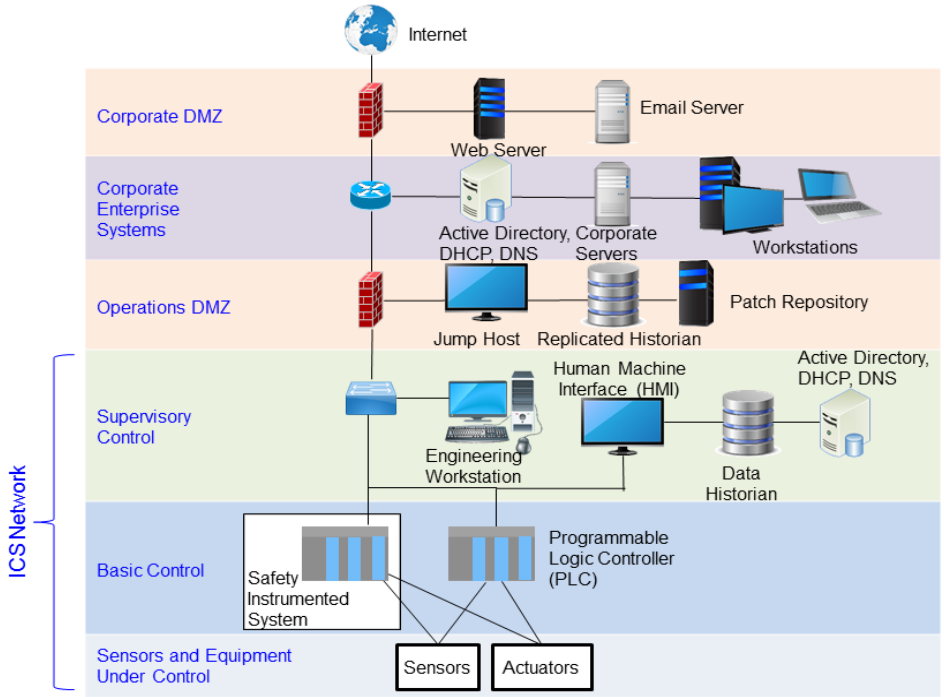

Fig. 1 Notional corporate and ICS networks

The Modbus protocol is often used for communications between an HMI and a PLC. A PLC is an electronic device that controls machinery and receives inputs from physical-plant sensors and actuators. An HMI is a software application that polls a PLC frequently for status information and sends instructions to PLCs to modify the behavior of the ICS. A human plant operator monitors the HMI for situational awareness about the ICS. HMIs may also provide a capability for the human operator to manually control a process, if needed.

A network intrusion-detection system (IDS) may detect suspicious network activity using behavior analysis or signature-based methods. The US Army Research Laboratory's (ARL's) Army Cyber-Research Analytics Laboratory created the ICS process-oriented intrusion detection (iPoid) algorithm. It is useful in protecting an ICS by monitoring critical-process variables. We define "critical-process variables" as those a plant operator deems the most critical for operation.[2] Critical-process variables and their severity should be defined cooperatively by security analysts and plant operators. We define a "plant operator" as an ICS professional who monitors the automation processes and knows the nominal values and low/high thresholds of critical-process variables. Baselining automation processes during quiescent operations will aid in crafting rules.

In this report, we describe how to use the ARL iPoid software to inspect Modbus Transmission Control Protocol (TCP) packets for anomalies and how to write inspection rules. To test iPoid, we simulated a PLC using software (see the Appendix for the unit test procedures).

2. iPoid Modbus Packet-Inspection Capability

The iPoid inspects Modbus TCP packets and creates alert messages for security analysts. In our algorithm, we define an "alert" as automatically generated information sent to a security analyst. The iPoid algorithm can be extended to inspect additional ICS protocols.

2.1 Software Requirements

The iPoid can be installed on an IDS sensor node running "Bro" software. The iPoid requires Bro Version 2.4.1 or later and Python 2 or later. The iPoid algorithm is implemented using 2 scripts: a Python script "ApplyRules.py" and a Bro script "PlcState.bro". In addition, a rules file is needed, which is written in the JavaScript Object Notation (JSON) format. We depict a sample sensor configuration in Fig. 2.

The PlcState.bro script creates a string of data in JSON format that represents the state of a PLC coil or register after a Modbus request–response message exchange

has completed. The ApplyRules.py script reads in rules written to create an alert message for a security analyst to investigate an intrusion with support from plant operators. The rule-checking process is performed outside Bro to enable users to dynamically update the inspection rules without restarting Bro. This avoids a risk of Bro missing packets on the wire.

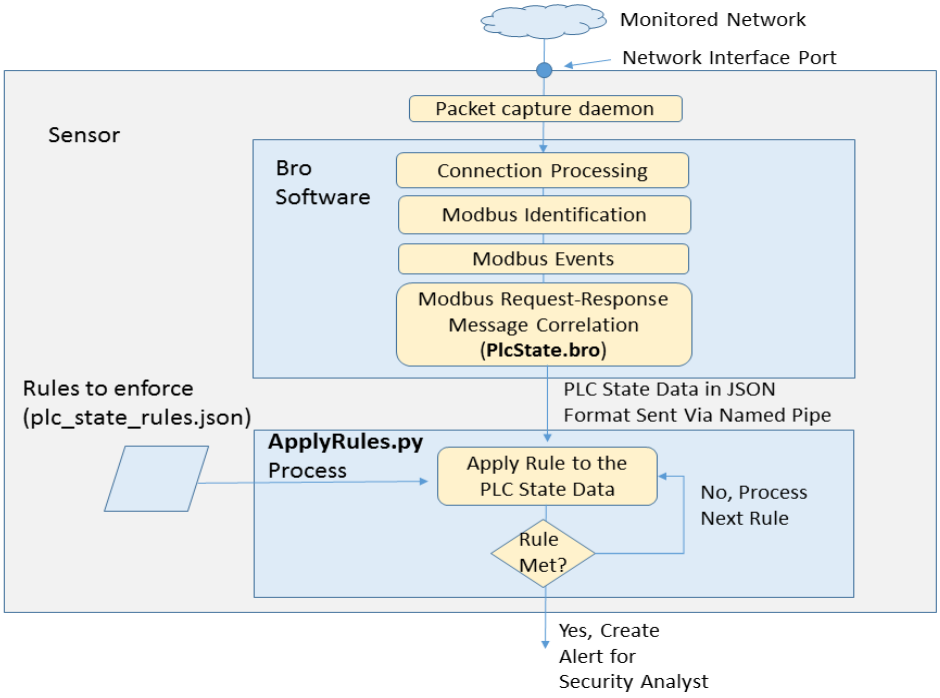

Fig. 2 iPoid components on a sensor

2.2 Startup Dependencies

The ApplyRules.py script must be started prior to Bro. At startup, the ApplyRules.py process will establish a named pipe to receive the Modbus data in JSON format from the PlcState.bro script executed by Bro. The ApplyRules.py script is started with the name of the rules file in the command line. After the ApplyRules.py process is running, then Bro can be started with the PlcState.bro script. Bro is typically configured to launch scripts listed in the local.bro file. Bro can also run the PlcState.bro script as a command-line argument.

2.3 iPoid Rules Format

The rules are leveraged by iPoid to ensure the current PLC state, inferred from Modbus traffic, is within normal operating conditions.

Two categories of rules may be specified. Function rules watch for Modbus function codes that reveal actions within the ICS and PLCs. Value rules monitor

the values of registers or coils, which is the current system state of the ICS and PLCs. A rules file may contain a block of one or both types of rules. For example, a plant IDS may only use value rules to monitor critical-process variables. However, the file may not have multiple blocks of the same rule type.

A rule file is written in JSON format; Fig. 3 diagrams the structure.

```
{ "FunctionRules":
    [ { Individual Function Rule },
      { Individual Function Rule },
      { Individual Function Rule },
        . . .
      { last Individual Function Rule } ],
  "ValueRules":
    [ { Individual Value Rule  },
      { Individual Value Rule  },
      { Individual Value Rule  },
        . . .
      { last Individual Value Rule } ]
}
```

Fig. 3 JSON rule file's structure

A rule file begins and ends with a brace. A list of rules begins either with the keyword "FunctionRules" or "ValueRules" followed by a comma-separated list of the respective rule type.

Each individual rule in Fig. 3 is written in the Backus–Naur Format (BNF) notation of {<condition>, <action>} whose construction is described in Sections 2.3.1 and 2.3.2. In both types of rules, the relative address of the register or coil is needed. If "Ignore" is written anywhere in a rule (e.g., for testing), the rule will not be evaluated.

A rule is evaluated as either "TRUE" or "FALSE" when applied to a Modbus transaction. If the rule evaluates to TRUE, then an alert message is created from the text in the <action> section of a rule.

A rule <action> field begins with the keyword "Message" and concludes with the text message that will be the alert text. The severity (Critical, High, etc.) should be included in the alert message.

2.3.1 Individual Function-Rule Format

A function rule may be used to alert the security analyst if a threat actor is enumerating a PLC or is sending write commands to change setpoints, which can

affect production. This rule type checks if a user-specified Modbus function code is applied to a register or coil. Figure 4 illustrates the rule structure.

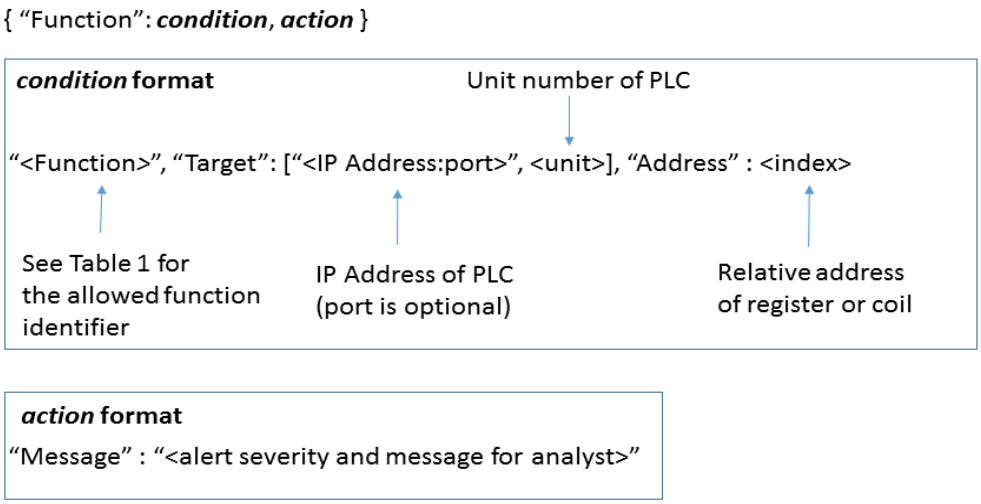

Fig. 4 Individual function-rule structure: rule condition consists of the Modbus function code, PLC IP address, and relative coil or register address

The function-rule *condition* field begins with a Modbus function identifier followed by 2 pairs of keywords and values to identify the PLC and data address to apply the rule to. Table 1 lists each Modbus function code supported by iPoid along with the data type, address range, and Modbus function identifier for the rule.

Table 1 Modbus function codes, data types, and address ranges capable of being monitored by iPoid

Modbus function code	Description	Data width	Address range	<Function> identifier in rule
1	Read coils	1 bit	00001 – 10000	READ_COILS
3	Read multiple holding registers	16 bits	40001 – 50000	READ_HOLDING_REGISTER
4	Read input register	16 bits	30001 – 40000	READ_INPUT_REGISTER
5	Write single coil	1 bit	00001 – 10000	WRITE_COILS
6	Write single holding register	16 bits	40001 – 50000	WRITE_HOLDING_REGISTER
15	Write multiple coils	16 bits	00001 – 10000	WRITE_COILS
16	Write multiple holding registers	16 bits	40001 – 50000	WRITE_HOLDING_REGISTER

After the Modbus function identifier is the PLC IP address and unit number to be monitored. The next field specifies the Modbus address of interest. This is the end

of the *condition* section of a function rule. (The action format of the rule was described previously in Section 2.3.)

Table 2 gives a few examples of function rules and describes when an alert message is created for each.

Table 2 Example Modbus function rules

Rule	Description
{"Function": "READ_HOLDING_REGISTER", "Target": ["192.168.200.13", 1], "Address": 5, "Message": "Severity: Low. Read Holding Register 40006 at 192.168.200.13.1"}	An alert will be created when PLC Unit 1 with IP address 192.168.200.13 has holding register address 40006 polled. The alert will contain the content of the "Message" field.
{"Function": "WRITE_HOLDING_REGISTER", "Target": ["192.168.200.13", 1], "Address": 3, "Message": "Severity: High. Write Holding Register 40004 at 192.168.200.13.1"}	An alert will be sent when PLC Unit 1 with IP address 192.168.200.13 has data written to holding register address 40004.
{"Function": "READ_COIL", "Target": [192.168.200.13", 1], "Address": 3, "Message": "Severity: Low. Read Coil 00004"}	An alert will be generated when PLC Unit 1 with IP address 192.168.200.13 has a coil read at address 00004.

2.3.2 Individual Value-Rule Format

A value rule enables iPoid to alert the security analyst when a register or coil value changes or if a register value is equal to, not equal to, less than, or greater than a setpoint. In addition, a rule can be written to create an alert if a register value changes by an amount (i.e., exceeds a delta). A value rule is useful to alert a security analyst if a cyber attacker is trying to change the PLC configuration or if the ICS is in a dangerous state. Rules can be written to alert if a critical-process variable exceeds upper or lower limits. The ApplyRules.py process creates a state table of each register and coil based upon the data in Modbus messages. When assessing if a register or coil has changed, the current value parsed from the most recent message is compared to the previous value contained in the ApplyRules.py process state table. Figure 5 illustrates the structure of <condition> and <action> fields for value rules.

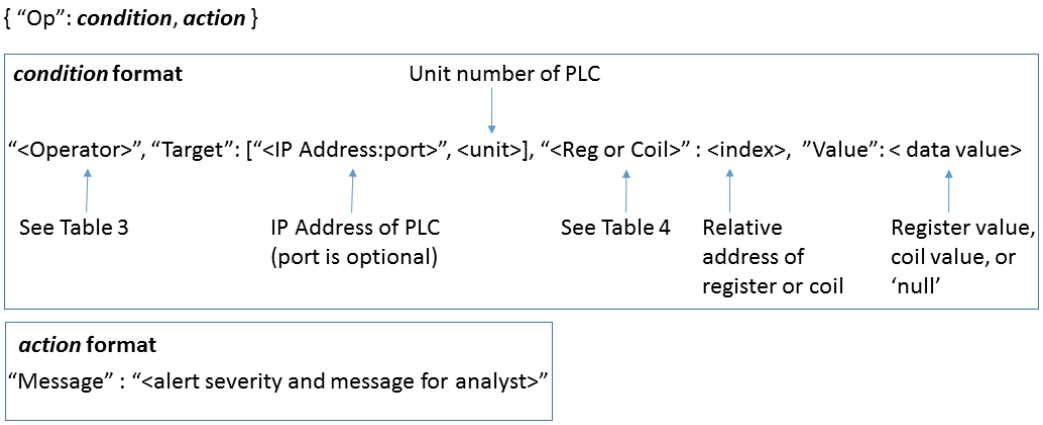

Fig. 5 Coil or register value-rule structure is different from a function rule because it has a comparison operator and setpoint to compare to the current coil or register value.

A value rule begins with the keyword "Op" followed by the *condition* expression. The <Operator> in Fig. 5 specifies the relationship to apply when comparing the current PLC holding register or coil value to the value in the rule. Each relationship operator has several ways to be designated (see Table 3). The text for the relationship operator is not case sensitive.

Table 3 PLC value-rule relationship–operator syntax

Relationship between value in rule to PLC value	Allowed <Operator> identifier in rule
PLC data IS EQUAL to value in rule	Equal, Equals, =, ==
PLC data IS NOT EQUAL to value in rule	NotEqual, NotEquals, !=
PLC data is GREATER THAN value in rule	GreaterThan, Greater, >
PLC data is LESS THAN PLC value in rule	LessThan, Less, <
PLC data CHANGED from value in rule	Changes, Change

Following the relationship identifier is the "Target" keyword with IP address and unit number of the PLC. The next field specifies the data type to be compared (coil or register); see Table 4 for allowed values.

Table 4 PLC value rule's data-type identifiers

Modbus data type	Data length	<Reg or Coil> field in rule
Holding register	16 bits	HoldingRegister
Input register	16 bits	InputRegister
Discrete output (coil)	1 bit	Coils

The next 2 fields are optional. If the rule specifies a value (e.g., threshold) to compare against the coil or register value, include a "Value" keyword and then the value to be compared. For a coil, this is 1 (TRUE) or 0 (FALSE) or a numeric value for a register.

To create an alert if a register value has changed by a certain amount (delta), include the "ChangeOp" keyword and then the relationship operator (see Table 3, second column) to apply when comparing the current register value to the one in the rule. When a rule with "ChangeOp" is evaluated, the previous PLC register value from the ApplyRules.py state table is compared to the current value and an alert message is created if the difference meets the conditions in the rule. Figure 6 illustrates the structure of a value rule to perform a delta comparison.

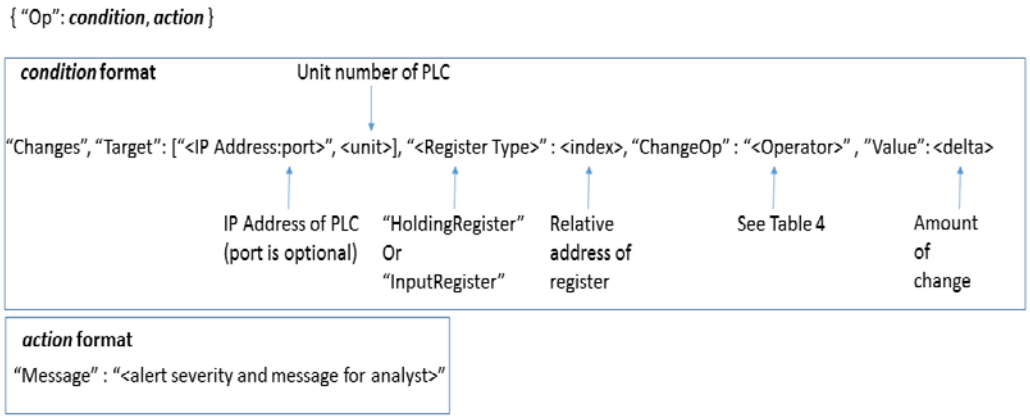

Fig. 6 Value-rule structure to detect specified differences in register state: Use "Changes" as the <Operator> and the "ChangeOp" keyword with the delta value.

Table 5 has several examples of Modbus value rules and describes the corresponding alert criterion.

Table 5 Example Modbus value rules

Value rule	Description
{"Op": "Changes", "Target": ["192.168.200.13", 1], "HoldingRegister": "0", "Message": "Severity: High. Holding Register 40001 changed"}	An alert will be created when PLC Unit 1 with IP address 192.168.200.13 has a change in holding register address 40001. The detect will contain the content of the "Message" field. The alert will be created for any change in the holding register.
{"Op": "Equals", "Target": ["192.168.200.13", 1], "Coils": 3, "Value": 0, "Message": "Severity: Critical. Valve is closed - Coil 00004 at 192.168.200.13.1"}	An alert will be sent when PLC Unit 1 with IP address 192.168.200.13 has a FALSE value at coil address 00004.
{"Op": "Changes", "Target": [192.168.200.13", 1], "InputRegister": 1, "ChangeOp", ">", "Value": 100, "Message": "Severity: High. Analog Input 30004 has changed greater than 100 psi"}	An alert will be generated when PLC Unit 1 with IP address 192.168.200.13 has a change of greater than 100 for input register at address 30004.

2.3.3 Updating Modbus Inspection Rules

The inspection rules may be updated without interruption to running processes. The ApplyRules.py process may be instructed to reload the inspection rules by a user sending a signal hangup (SIGHUP) message to the process. We illustrate the steps to update the rules in the following procedures:

Step 1: Update the inspection rules in a JSON-formatted text file. Step 2: Obtain the process identifier (ID) of the ApplyRules.py process.

```
$ ps aux | grep ApplyRules.py
```

The process ID will be returned with the command line that initiated the ApplyRules.py process.

Step 3: Send a SIGHUP command to the process ID of ApplyRules.py.

```
$ kill –HUP <process ID of ApplyRules.py>
```

You will see this message in the ApplyRules.py process-standard out window or file: `> Loaded new ruleset`

2.4 iPoid Alert-Message Format

The ApplyRules.py process will write an alert message to the operating system's (OS's) standard output when a rule is triggered (i.e., anomalous activity detected). This alert can also be piped to a file or to an IDS application programming interface (API) for additional processing and display.

As Fig. 7 shows, an alert consists of 2 rows of text. The first row presents information about the PLC and the second row is a message for the security analyst.

The structure of the alert first row is an "A" character followed by the IP address of the PLC, followed by the PLC unit number. Next is a "T" character for data type. After the "T" character may follow a "1" if the alert was caused by a holding register value; a "2" if the cause is an input register value; a "3" if this alert is due to a coil; or "0" if the data type is undefined.

An "O" character follows with the type of Modbus operation ("R" for read and "W" for write). Next is the timestamp (i.e., "TS") of the alert in epoch format. After the timestamp is a list of the current PLC coil and register values parsed from the Modbus transaction that triggered the alert.

The second row of the alert message is text from the <action> section of the rule. We recommend as much context as possible about the severity of the alert and impacted process and that a point of contact be included in the alert message. These details will assist the security analyst with their investigation.

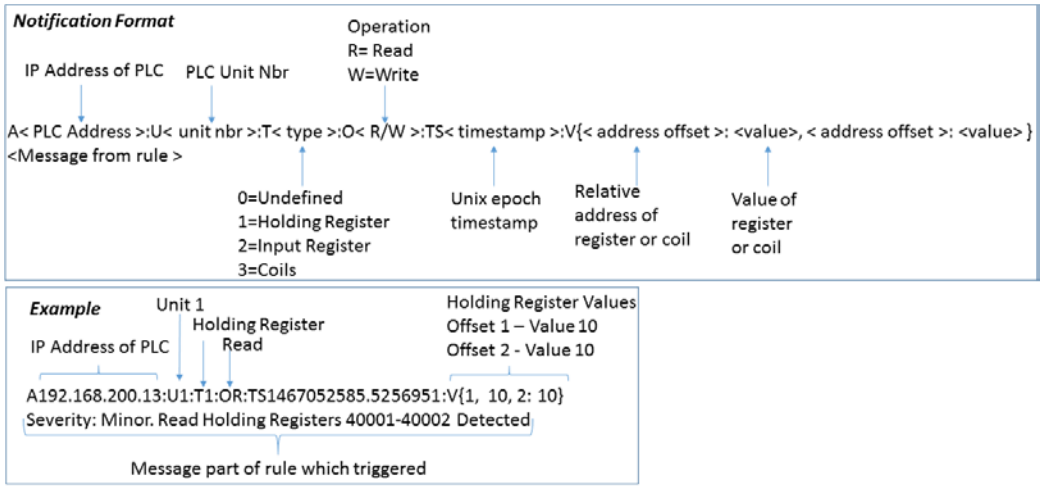

Fig. 7 iPoid alert-message format

3. Examples of iPoid Implementation

In this section we present 2 examples of how iPoid can be deployed with a research IDS to protect ICS networks. In Section 3.1 we demonstrate how iPoid can monitor critical-process variables. In Section 3.2 we explain how iPoid can detect an unallowed Modbus command sent to a PLC.

3.1 Protect Critical-Process Variables

Figure 8 illustrates part of a Meal, Ready-To-Eat (MRE) process where a research IDS and iPoid are monitoring critical-process variables.[3] The MRE plant produces high-quality meals for Soldiers. The meat and vegetables are cooked separately and once cooked, the meals are prepared and packaged using high-pressure processing to have a long shelf life. As depicted, a switched port analyzer (SPAN) port of each switch forwards network traffic to each sensor. Software within the IDS management servers connect to each sensor and retrieve the packet data.

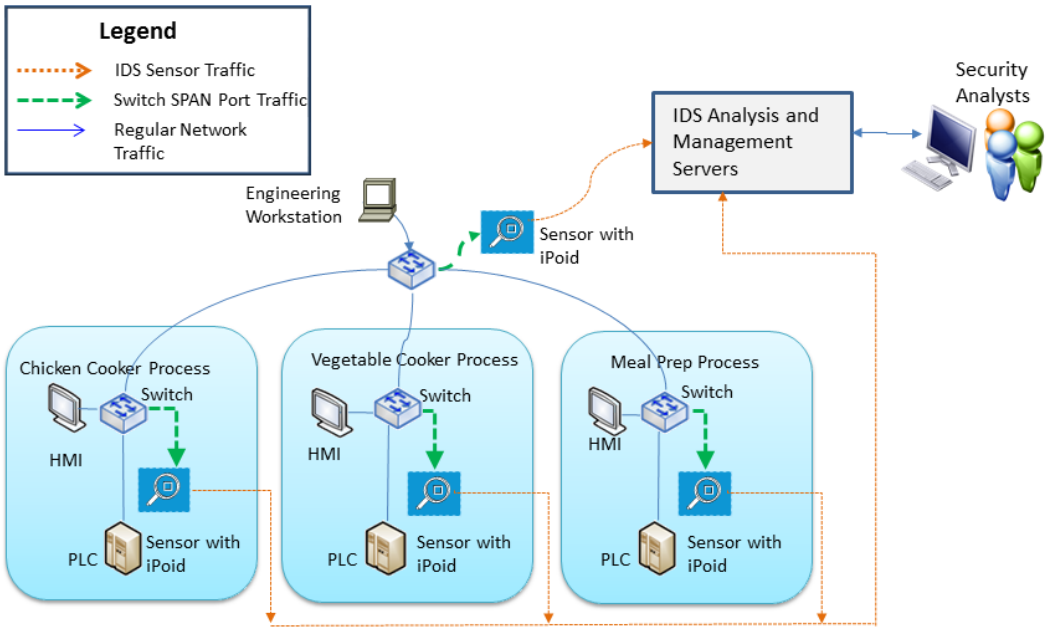

Fig. 8 Example deployment to monitor critical-process variables

The plant operator has identified all critical-process variables and one of the critical variables is the chicken cooker's temperature. The oven temperature must be above 350 °F during production operations, otherwise the chicken will be undercooked and pose health risks to Soldiers later consuming the MREs. The chicken-cooker PLC stores the oven temperature in a holding register and this value is reported to the HMI every 30 s during routine polling. Table 6 illustrates a rule in the JSON format to create an alert message if the Modbus holding-register value for the oven temperature is below 350 °F.

Table 6 Example of value rule to alert when oven's holding-register value is below 350 °F

Value rule	Description
{"Op": "LessThan", "Target": ["192.168.200.13", 1], "HoldingRegister": "0", "Value": 350, "Message": "Severity: High. Oven temp setpoint less than 350F"}	An alert will be created when the chicken cooker's PLC (IP address 192.168.200.13) has a temperature (stored in holding register with offset 0 [address 40001]) is below 350 °F. The alert will contain the content of the "Message" field.

3.2 Protect Against Unallowed Modbus Commands

Figure 9 portrays the same MRE manufacturing process with a data historian in an additional Operations Support enclave polling PLC-state data from each HMI. In this topology, the plant operator has designated allowed Modbus function codes for each enclave. The plant operator specified that hosts in the Operations Support enclave should not send Modbus write commands to a PLC.

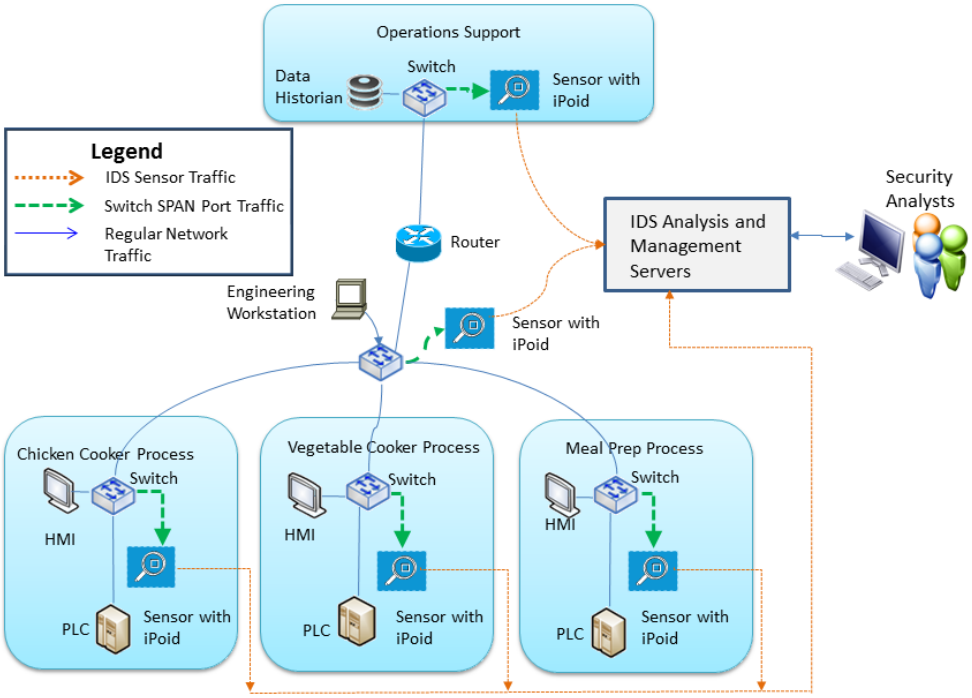

Fig. 9 ICS network with data historian

Table 7 illustrates a rule for the sensor monitoring the Operations Support enclave to create an alert message if a Modbus write command is sent to a PLC.

Table 7 Example of rule to detect an unallowed Modbus command

Value rule	Description
{"Function": "WRITE_HOLDING_REGISTER", "Target": ["192.168.200.13", 1], "Address": "0", "Message": "Severity: Critical. Unauthorized Write Command Sent to PLC Holding Register 40001"}	An alert will be created when a host in the Operations Support enclave sends a Modbus holding register write command (function Code 6) to the chicken cooker's PLC (IP address 192.168.200.13) for address 40001.

4. Conclusion and Discussion

The ARL iPoid's inspection capability is flexible and can be customized to monitor any ICS network using the Modbus TCP protocol. The iPoid can check for unallowed Modbus commands sent to a PLC and verify critical-process variables are within safe limits. These functions are not available in corporate IDSs. We recommend security analysts and plant operators collaboratively write the iPoid rules. Each rule should include a severity level as well as contextual information to assist security analysts in responding to an alert. ICS networks rarely change in order to maintain high availability, so any alert created by iPoid may indicate an intrusion has occurred.

5. References

1. Colbert EJM, Kott A, editors. Cyber-security of SCADA and other industrial control systems. New York (NY): Springer; 2016.

2. Colbert E, Sullivan D, Hutchinson S, Renard K, Smith S. A process-oriented intrusion detection method for industrial control systems. Paper presented at: ICCWS 2016. Proceedings of the 11th International Conference on Cyber Warfare and Security; 2016 Mar 17–18; Boston (MA).

3. Sullivan DT, Colbert EJ. Demonstration of supervisory control and data acquisition (SCADA) virtualization capability in the US Army Research Laboratory (ARL)/Sustaining Base Network Assurance Branch (SBNAB) US Army Cyber Analytics Laboratory (ACAL) SCADA hardware testbed. Adelphi Laboratory Center (MD): Army Research Laboratory (US); 2015 May. Report No.: ARL-CR-0773.

Appendix. Unit Test Procedures

This appendix describes how a tester or security analyst can exercise industrial control system process-oriented intrusion detection (iPoid) and test inspection rules using an intrusion-detection system (IDS) sensor, a simulated Programmable Logic Controller (PLC), and a Modbus client operating on virtual machines (VMs) in a test lab. The sensor must be capable of running Bro 2.4.1, or later. (At the time of this writing, Bro can only run on hosts with UNIX[1] operating systems.)

A.1 Test Setup

Several components are required and these can be hosted on VMs in a lab environment. To conduct unit testing, below are the required components which can be virtualized with software such as VirtualBox[2]:

- Sensor with Bro 2.4.1 and Python 2.7.
- Modbus TCP server to simulate a PLC. We used ModSim32[3] to simulate a PLC because this tool simulates coils, contacts, holding registers, and input registers.
- Modbus TCP client such as Simply Modbus TCP Client.[4] This tool can craft messages for all Modbus functions that can be inspected by iPoid.

We present an example of the network topology to test the Modbus packet inspection capability in Fig. A-1. In this unit test network, the tester can create transactions of different function codes to poll a simulated PLC using Modbus TCP. The tester can set values for coils, input, and holding registers in ModSim32 as a real PLC would have.

[1] UNIX is a registered trade mark in the United States and other countries, licensed exclusively through X/Open Company Limited.
[2] VirtualBox. Version 5.0.18. Redwood Shores (CA); Oracle Corporation; [accessed 2016 May 16]. https://www.virtualbox.org.
[3] ModSim32. Version 4.A00-04. Lewisburg (WV): WinTECH Software Design; [accessed 2016 June 27]. http://www.win-tech.com/html/modsim32.htm.
[4] Simply Modbus TCP Client. Version 1.4. Simply Modbus; [accessed 2016 June 27]. http://www.simplymodbus.ca/TCPclient.htm.

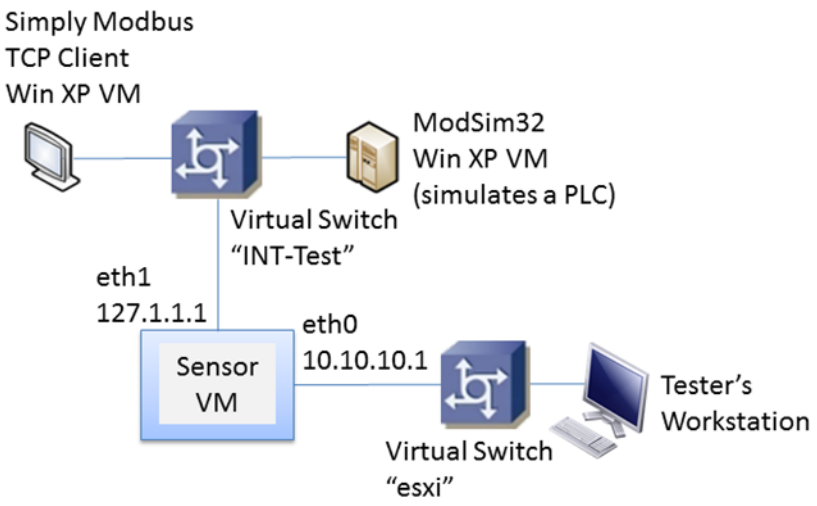

Fig. A-1 Unit test network

Table A-1 lists each virtualized host of the test network, the IP address of each interface, and which virtual switch an interface connects to.

Table A-1 Test network configuration

Virtual machine	Interface	Virtual switch name	IP address
Tester's workstation	eth0	esxi	10.10.0.1
Sensor	eth0	esxi	10.10.10.1
	eth1	INT-Test	127.1.1.1
Simply Modbus TCP client	eth0	INT-Test	192.168.210.14
ModSim32 (simulate a PLC)	eth0	INT-Test	192.168.210.13

In this test network, interface eth1 of the sensor functions as the packet-sniffing interface and is connected to the INT-Test virtual switch. The sensor's VM network configuration has eth1 configured for "Allow All" for the Promiscuous Mode setting in VirtualBox to enable packet sniffing to work.

A.2 Test Procedures

We present sample test cases to verify the Modbus packet-inspection software is working correctly. Each test case is written to detect a condition when the PLC is in a state out of compliance with normal plant operations. Since this is unit testing on the sensor, we will present the alert-message output from the ApplyRules.py script when a rule to detect an anomalous condition is met. This message can be sent to the IDS graphical user interface (GUI) via the IDS application programming interface (API). However, for unit testing, seeing the alert message printed to the operating system's (OS's) standard output is sufficient to validate iPoid is working.

The sample test cases are the following:

1) Unallowed Modbus function code sent to a holding register.
2) Coil value has changed to an unallowed value.
3) Holding register is below a setpoint.
4) Input register value changed above a threshold.

We depict a separate rules file for each case to codify the conditions when the PLC state, inferred by the Modbus messages, is not according to plant-design parameters.

A.2.1 How to Run Each Test Case

For unit testing, we recommend the tester use a separate Linux workstation with X-Windows and open remote-terminal windows to the sensor using secure shell (SSH) to run the test cases. Perform the following steps to run ApplyRules.py and Bro for unit testing:

Step 1. Open a SSH terminal window to the sensor and run ApplyRules.py with the rules file for each test case. The output of each test case will be displayed in this window by the OS standard out stream. This step will create a named pipe to receive Modbus transaction data from Bro.

```
$ python ApplyRules.py <Name of rules file in JSON format>
```

Example:

```
$ python ApplyRules.py plc_state_rules.json
```

Step 2. Open a separate SSH terminal window to the sensor. Elevate your privileges to the root user and start Bro with the PlcState.bro script. Root privileges are required in order for Bro to capture packets on interface eth1.

```
# /usr/local/bro/bin/bro –b –i eth1 ./PlcState.bro
```

A.2.2 Test Case 1

In Test Case 1, the ApplyRules.py process will create an alert message if a Modbus write command is sent to holding register 40003.

Step 1. Using a text editor, create a text file with name test-case-1-rule.json and enter this rule:

```
{“FunctionRules”:
```

```
      [{“Function”: “WRITE_HOLDING_REGISTER”, “Target”:
[“192.168.200.13”, 1], “Address”: 2, “Message”: “Severity:
Medium. Write Holding Register 40003 Detected”}]
}
```

Step 2. Start ApplyRules.py with the name of the rules file on the command line, then start Bro (as detailed in Section A.2.1).

Step 3. In ModSim32, start the server function by clicking on these menu commands: Connection-> Connect-> Modbus/TCP Svr.

Step 4. In ModSim32, set the Device Id to 1, Address to 0001, and Modbus Point Type to “03: Holding Register”. You will see a table of Modbus holding registers with their data values starting at address 40001.

Step 5. In the Simply Modbus TCP Client window, establish a TCP connection to ModSim32. Set the Slave ID to “1”. Press the “Write” button and a window will appear with parameters to enter to create a Modbus write message. Set the Function Code to “6”, Minus Offset to “0”, First Register to “1”, #Values to Write to “1”, and Register Size to “16 bit registers”. Set Values to Write to any number, then press the Send button.

Step 6. In the terminal window with the ApplyRules.py process you should **not** see an alert message. You should see the value you entered in the Simply Modbus TCP Client Write window appear in holding register 40002 in the ModSim32 window. This validates the rule did not trigger because the write command was not sent to address 40003.

Step 7. In the Simply Modbus TCP Client Write window, set the First Register to “2”, then press the Send button. This will send the write message to ModSim32 again.

Step 8. In the terminal window with the ApplyRules.py process you **should** see an alert message similar to this:

```
A192.168.200.13:U1:T1:OW:TS1467056889.7762811:V{2:10}
Severity: Medium. Write Holding Register 40003 Detected
```

A.2.3 Test Case 2

In Test Case 2, the ApplyRules.py process will create an alert message if a coil with address 00003 has a value of 0 (FALSE). This could be an example of a critical valve in a manufacturing process being closed.

Step 1. Using a text editor, create a text file with name test-case-2-rule.json and enter this rule:

```
{“ValueRules”:
       [{“Op”: “Equal”, “Target”: [“192.168.200.13”, 1],
“Coils”: “2”, “Value”: 0, “Message”: “Severity: Critical.
Valve at Address 00003 is Closed”}]
}
```

Step 2. Start ApplyRules.py with the name of the rules file on the command line, then start Bro (as detailed in Section A.2.1).

Step 3. In ModSim32, start the server function by clicking on these menu commands: Connection-> Connect-> Modbus/TCP Svr.

Step 4. In ModSim32, set the Device Id to “1”, Address to “0001”, and Modbus Point Type to “01: Coil Status”. You will see a table of Modbus coils with their data values starting at address 00001. Set the values of the first 5 coils to “1” (TRUE).

Step 5. In the Simply Modbus TCP Client window, establish a TCP connection to ModSim32. Set the Slave ID to “1”. Set the Function Code to “1”, Minus Offset to “0”, First Coil to “0”, No. of Coils to “5”, and Register Size to “1 bit coils”, then press the Send button.

Step 6. In the terminal window with the ApplyRules.py process you should **not** see an alert message. This validates the rule did not trigger because coil address 00003 is “1” (TRUE).

Step 7. In ModSim32, set the value of coil 00003 to “0” (FALSE).

Step 8. In the Simply Modbus TCP Client window, press the Send button. This will poll ModSim32 again.

Step 9. In the terminal window with the ApplyRules.py process you **should** see an alert message similar to this:

```
A192.168.200.13:U1:T3:OR:TS1467058805.2187691:V{0: True, 1:
True, 2: False, 3: True, 4: True}
Severity: Critical. Valve At Address 00003 is closed
```

A.2.4 Test Case 3

In Test Case 3, the ApplyRules.py process will create an alert message if a holding register value is above a setpoint. Holding register 40010 will be above 50000 psi, which could be an extreme pressure in a storage container.

Step 1. Using a text editor, create a text file with name test-case-3-rule.json and enter this rule:

```
{“ValueRules”:
       [{“Op”: “GreaterThan”, “Target”: [“192.168.200.13”,
1], “HoldingRegister”: “9”, “Value”: 50000, “Message”:
“Severity: High. Tank pressure at Address 40010 is above
50000 psi”}]
}
```

Step 2. Start ApplyRules.py with the name of the rules file on the command line, then start Bro (as detailed in Section A.2.1).

Step 3. In ModSim32, start the server function by clicking on these menu commands: Connection-> Connect-> Modbus/TCP Svr.

Step 4. In ModSim32, set the Device Id to 1, Address to 0001, and Modbus Point Type to “03: Holding Register”. You will see a table of Modbus holding registers with their data values starting at address 00001. Set the value of holding register 40010 to 500.

Step 5. In the Simply Modbus TCP Client window, establish a TCP connection to ModSim32. Set the Slave ID to “1”. Set the Function Code to “3”, Minus Offset to “0”, First Register to “9”, No. of Regs to “1”, and Register Size to “16 bit registers”, then press the Send button.

Step 6. In the terminal window with the ApplyRules.py process you should **not** see an alert message. This validates the rule did not trigger because the holding register at address 40010 does not have a value above 50000.

Step 7. In ModSim32, set the value of holding register 40010 to “50001”.

Step 8. In the Simply Modbus TCP Client window, press the Send button. This will poll ModSim32 again.

Step 9. In the terminal window with the ApplyRules.py process you **should** see an alert message similar to this:

```
A192.168.200.13:U1:T1:OR:TS1467117361.855993:V{9: 50001}
Severity: High. Tank pressure at Address 40010 is above
50000 psi
```

A.2.5 Test Case 4

In Test Case 4, the ApplyRules.py process will create an alert message if an input register value changes more than a set amount. This simulates an analog pressure sensor reporting a sudden change of more than 1000 psi in a storage tank.

Step 1. Using a text editor, create a text file with name test-case-4-rule.json and enter this rule:

```
{“ValueRules”:
       [{“Op”: “Changes”, “Target”: [“192.168.200.13”, 1],
“InputRegister”: “9”, “ChangeOp” : “>”, “Value”: 1000,
“Message”: “Severity: Medium. Tank pressure at Address
30010 has changed more than 1000 psi”}]
}
```

Step 2. Start ApplyRules.py with the name of the rules file on the command line, then start Bro (as detailed in Section A.2.1).

Step 3. In ModSim32, start the server function by clicking on these menu commands: Connection-> Connect-> Modbus/TCP Svr.

Step 4. In ModSim32, set the Device Id to 1, Address to 0001, and Modbus Point Type to “04: Input Register”. You will see a table of Modbus input registers with their data values starting at address 00001. Set the value of holding register 30010 to 500.

Step 5. In the Simply Modbus TCP Client window, establish a TCP connection to ModSim32. Set the Slave ID to “1”. Set the Function Code to “4”, Minus Offset to “0”, First Register to “9”, No. of Regs to “1”, and Register Size to “16 bit registers”, then press the Send button. The ApplyRules.py process will create a state table for the PLC with the value of 500 assigned to input register 30010.

Step 6. In ModSim32, set the value of holding register 30010 to “1500”.

Step 7. In the terminal window with the ApplyRules.py process you should **not** see an alert message. This validates the rule did not trigger because the amount of change of input register at address 30010 did not exceed 1000.

Step 8. In ModSim32, set the value of holding register 30010 to “1501”.

Step 9. In the Simply Modbus TCP Client window, press the Send button. This will poll ModSim32 again.

Step 10. In the terminal window with the ApplyRules.py process you **should** see an alert message similar to this:

```
A192.168.200.13:U1:T2:OR:TS1467120811.7215459:V{9: 1501}
Severity: Medium. Tank pressure at Address 30010 has
changed more than 1000 psi
```

List of Symbols, Abbreviations, and Acronyms

ACAL	Army Cyber-Research and Analytics Laboratory
API	application programming interface
ARL	US Army Research Laboratory
BNF	Backus–Naur Format
DHCP	Dynamic Host Configuration Protocol
DMZ	demilitarized zone
DNS	Domain Name System
DOD	Department of Defense
GUI	graphical user interface
HMI	human–machine interface
ICS	industrial control system
ID	identifier
IDS	intrusion-detection system
IP	Internet Protocol
iPoid	ICS process-oriented intrusion detection
IT	information technology
JSON	JavaScript Object Notation
MRE	Meal, Ready-To-Eat
OS	operating system
PLC	Programmable Logic Controller
SCADA	supervisory control and data acquisition
SIGHUP	signal hangup
SPAN	Switched Port Analyzer
SSH	secure shell
TCP	Transmission Control Protocol

VM	virtual machine

1 DEFENSE TECH INFO CTR
(PDF) DTIC OCA

2 US ARMY RSRCH LAB
(PDF) IMAL HRA MAIL & RECORDS MGMT
RDRL CIO L TECHL LIB

1 GOVT PRNTG OFC
(PDF) A MALHOTRA

12 US ARMY RSRCH LAB
(PDF) RDRL CIH C
J CLARKE
K RENARD
RDRL CIN
A KOTT
N VALLESTERO
RDRL CIN D
B RESCHLY
E COLBERT
T PARKER
RDRL CIN S
JW SCHAUM
C ARNOLD
C WALSH
D SULLIVAN
P TUCKER

INTENTIONALLY LEFT BLANK.